Themis: Economy-Based Automatic Resource Scaling for Cloud Systems

Stefania Victoria Costache, Nikos Parlavantzas, Christine Morin, Samuel Kortas

Themis: Economy-Based Automatic Resource Scaling for Cloud Systems

Stefania Costache *†, Nikos Parlavantzas ‡, Christine Morin †, Samuel Kortas *
*EDF R&D, Clamart, France
†INRIA Rennes - Bretagne Atlantique Centre, Rennes, France
‡INSA, Rennes, France

***Abstract*—High performance computing (HPC) infrastructures are used to execute increasingly complex scientific applications with time-varying resource requirements. A key challenge for infrastructure providers is to distribute resources to such applications so that application performance objectives are met while guaranteeing a high infrastructure utilization. Most of existing solutions provide limited support for meeting application objectives and can lead to inefficient resource usage. In this paper we propose Themis, a system that uses an economic-based approach to automatically allocate resources to the applications that need them the most. Themis relies on a proportional-share auction that ensures fair differentiation between applications while maximizing the resource utilization. To support meeting application performance objectives, Themis provides generic scaling policies based on feedback control loops. These policies adapt the resource demand to the current infrastructure conditions and can be extended for different application types. We have evaluated the performance of the system through simulations, and the results show that Themis can effectively meet application performance objectives while optimizing infrastructure's utilization.**

I. Introduction

High performance computing (HPC) infrastructures are used to execute increasingly complex, dynamic scientific applications, such as scientific workflows and master-worker frameworks. These applications require or can efficiently use time-varying amounts of underlying resources, such as computing nodes, and have to meet various service level objectives (SLOs), such as respecting time constraints. A key challenge for infrastructure providers is to distribute resources to such dynamic applications so that their objectives are met while guaranteeing a high infrastructure utilization. Traditional resource management systems [16] fail to address this challenge. These systems provide a primitive interface for resource control, forcing users to overprovision resources to meet application SLOs, and having a negative impact on utilization.

In contrast to traditional HPC infrastructures, cloud infrastructures support a dynamic provisioning model in which users obtain virtualised resources *on-demand* and are charged according to a given pricing scheme [1]. This model is attractive for executing dynamic applications for several reasons. First, users can obtain resources "instantly", removing any need for over-provisioning. Second, the pricing scheme gives users the incentive to regulate their resource usage, allowing resources to be allocated to users that need them the most. Third, virtualisation gives providers the flexibility to increase the utilization of hardware resources. However, typical cloud resource management approaches (e.g., Amazon EC2 on-demand instances) suffer from the following limitation. User requests are either fully fulfilled or rejected, forcing the provider to reserve a sufficient amount of idle resources and thus limiting the infrastructure utilization.

An approach that overcomes the previous limitation is using a spot market with a dynamic pricing scheme. In this approach, users place bids for resources, and resource allocations and prices fluctuate depending on the overall demand. When the demand is low, the prices drop and users can request more resources; when the demand is high, prices increase and users are encouraged to request less resources. To apply this approach in managing HPC infrastructures, two issues must be addressed: (i) providing suitable spot market-based mechanisms that distribute virtual resources in a fair and efficient way, and (ii) supporting applications to meet their SLOs while using these mechanisms.

In this paper we present Themis[1], a system that applies a spot market to efficiently manage resources and meet application SLOs. Themis provides: (i) a *proportional-share auction* based scheduler; (ii) support for building feedback control loops that adapt the application bid and resource demand in order to meet application SLOs. The proportional-share auction ensures a fair use of resources while its fine-grained work-conserving nature maximizes the resource utilization. The feedback control loops are based on generic scaling policies, extended to target specific application types. Policies for two application types are currently provided, namely, for rigid and elastic applications. Simulation experiments show that our system is capable of providing effective support for application SLOs and fairness.

The rest of the paper is organized as follows. Section II introduces the motivations of this work. Section III details the design of our system and Section IV presents the experimental results and discusses the limitations of our solution. Finally, Section V concludes the paper.

II. Motivation and Related Work

In this section we detail the motivation behind our work and the research related to our goals.

[1]This work is supported by ANRT through the CIFRE sponsorship No. 0332/2010

A. Motivation

A general concern in managing HPC infrastructures is how to use all the available capacity to ensure the best application performance, given the workload heterogeneity. Applications executed on these infrastructures can be classified in two general classes: rigid and elastic. A rigid application can use a fixed number of virtual machines during its life-time. An example of a rigid application is an HPC application composed of a fixed set of tasks that need to be executed in parallel. An elastic application can use a dynamic number of virtual machines during its life-time. This number is dependent of the application's goal and the infrastructure resource availability. A master-worker framework can be such an application: a master component submits tasks to worker components that can be added or removed during the application's execution. Both types of applications are highly popular in the scientific community.

Applying the cloud "on-demand" provisioning model on these infrastructures can be beneficial for both users and providers. Elastic applications can scale dynamically and use as many resources as needed at any point. Moreover, as virtualization provides more flexibility to manage resources, fine-grained resource allocation models can be used to maximize the resource utilization. HPC rigid applications can take advantage of these models as they can be executed with fractional amounts of resources, thus avoiding waiting times in queues when there are idle resources. As a result, users can better meet their SLOs while the providers maximize their infrastructure resource utilization. In this context the main challenge that needs to be addressed is what provisioning models to implement and how users can meet their SLOs by using them.

Spot markets can be used as an efficient and fair resource allocation mechanism. With this mechanism resources are "instantly" allocated to users who value them the most while the provider benefits from the mechanism's property to automatically balance the user demand with the infrastructure's resource availability. Spot markets provide dynamic pricing which acts as a generic decentralized allocation mechanism: the resource allocation is not only the task of the resource provider, but it becomes a task distributed between applications/users and the infrastructure. By using this mechanism as a basis for allocating resources, the resource management becomes flexible and generic, allowing more application SLO types to be met.

Our goal is to *allocate fine-grained resources in an efficient and fair way by using a spot-market model while providing a generic support for application SLOs*.

B. Related Work

We group the related works in two categories: (i) works that focus on providing spot-market mechanisms; (ii) works that focus on providing support for application SLO while acquiring resources from a spot market.

Spot Market Mechanisms: Spot markets are usually implemented through auction mechanisms. In an auction users bid for resources and the provider assigns resources to the users with the highest bids. Recently, Amazon EC2 introduced Spot Instances [7], virtual machines sold through an auction, to sell unused resource capacity and increase the utilization of its infrastructure. To provision spot instances users submit a bid, representing the maximum price they are willing to pay. The spot instances are then available for the user as long as the spot instance price remains below the user's bid. When the price goes above user's bid, the instances are forcibly killed. Many grid resource allocation systems implemented auction protocols. Belagio [3] and Mirage [5] use a combinatorial auction to distribute bundles of distributed resources to users. The same type of auction was proposed in the context of cloud computing to provision classes of virtual machines [19]. These auction types are computationally expensive and to reduce the mechanism complexity users are forced to express their requirements in a coarse-grained manner (e.g., through virtual machine classes).

To overcome this drawback, we use in our work a proportional-share auction to achieve a fine-grained dynamic allocation of resources. This mechanism combines the good properties of the "classic" proportional-share allocation scheme with the efficiency brought by auctions. Through a proportional-share auction, each user submits a bid b_i, expressing its payment and receives a share $b_i/\Sigma b$ of the resource capacity C. The resource price is given by $\Sigma b/C$.

Even when the users are price anticipating (i.e., they anticipate the effect of changing their bids on the price), this mechanism achieves a good efficiency [6]. Moreover, the mechanism is work-conserving: there are no applications waiting for resources if some fractional amounts are available for use, leading to a maximum utilization of the infrastructure.

Several systems use a proportional-share auction for resource management. In Tycoon [8] each host runs a proportional-share auction while agents use a "Best Response" algorithm to select the hosts to which they submit their bids. A dynamic priority scheduler is proposed for Hadoop [13], to allocate map/reduce slots to an user's job. However, these systems do not support SLO management on top of proportional-share auctions. Libra [14] applies a proportional-share scheme on each host to allocate CPU proportional to the application's deadline while a global pricing mechanism is used to balance supply with demand. In contrast with this work, our approach is more flexible as we don't send the application SLO to the resource manager and we allow each agent to adapt itself to the changing resource prices.

SLA-ensurance Mechanisms: Earlier work addressed the problem of providing guidance to the user regarding how much to bid to meet its desired QoS given the resource price volatility [12]. This work was used to provide resource-level QoS guarantees and can be extended with application-level adaptation strategies.

More recent work focused on improving application execution when using spot instances. The main problem that comes from using this model is the lost computation time due to the unpredictable instance termination in case of "out-of-bid"

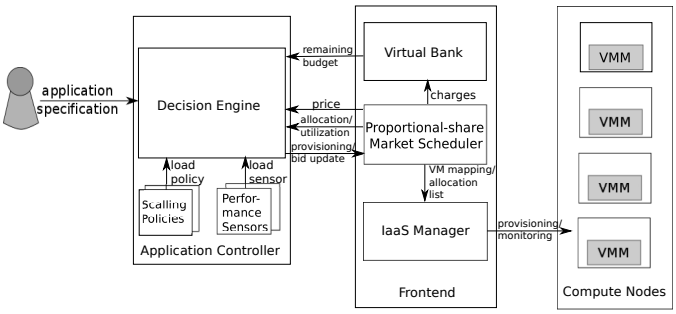


Fig. 1: System overview.

events. Andrzejak et al. use the spot price history and devise checkpoint strategies to minimize the application execution time[18]. Mazzucco et al. devise allocation policies to ensure the SLA for a web service provisioned with the same model [10]. Mattess et al. use the spot instances to manage peak loads on a cluster [9]. Our work is complementary. Our spot market is different from spot instances as our resource manager does not forcibly kill virtual machine instances but instead it diminishes the virtual machine's resource share. Our model gives applications more flexibility to react to price changes and we can leverage these approaches to design better policies.

III. Design

Figure 1 gives an overview of our system. Our system relies on the use of a virtual economy. Users receive amounts of currency from a virtual currency manager (i.e., a *Virtual Bank*), distributed based on predefined policies. To run their applications, users provision virtual clusters for which they purchase amounts of resources from an *market-based scheduler* that resides on the infrastructure's frontend. To allocate fractional amounts of resources on hosts, the market-based scheduler periodically runs a proportional-share auction. To ensure that each virtual machine receives its paid share and to maximize the resource utilization, the scheduler also performs load balancing. The operations regarding virtual machine management, networking and user management are performed by a Cloud IaaS manager (e.g., OpenNebula [15]).

As it is cumbersome for the user to manually scale its virtual cluster whenever its application demands change, each application has an associated *application controller*. For each application type there is a specific application controller. The application controller monitors the application and uses *feedback-based control loops* to adapt the application's bid and resource demand to the price variations and user SLO.

A. Virtual Currency Management

To ensure a fair use of resources, users dispose of limited amounts of currency (i.e., credits). When an application needs to be executed, the user creates an account for it and transfers an initial budget of currency. To avoid situations in which this amount is depleted during the application execution, the user can specify a rate at which the application's budget is replenished. However, the accumulated budget cannot exceed the initial amount given by the user. More complex policies can be applied to prevent hoarding of credits but their investigation is outside the scope of this paper.

```
1:  Variables:
2:      vms    // List of current vm requests
3:      hosts    // The current cluster configuration
4:      CostMax    // Maximum solution cost (number of
        migrations)
5:
6:  ComputePlacement()
7:      solution_current = hosts
8:      solution_best = solution_current
9:      tabu_list = ∅
10:     while termination condition is not met do
11:         host = pick the most overloaded host
12:         vm = pick the less valuable vm from the host such
            that (vm,host) is not in tabu_list
13:         destination = pick less overloaded host
14:         apply migration(vm, destination) to
            solution_current
15:         add (vm, host) to tabu_list
16:         if σ(solution_current) < σ(solution_best) and
            cost(solution_current) < CostMax then
17:             solution_best = solution_current
18:     return solution_best
```

Fig. 2: VM placement algorithm.

B. Resource Allocation and Load-Balancing

We describe next the scheduling algorithm applied by the market-based scheduler. We assume a homogeneous infrastructure and in the following we address only the CPU resource allocation case. The algorithm periodically takes as input the new and existing provisioning requests and outputs the CPU allocation and the physical placement of each virtual machine. The algorithm executes in two phases: (i) it computes the best allocations the virtual machines can get at the user's bid; (ii) it performs load-balancing to ensure the computed allocations within an error bound while considering the virtual machine migration cost.

Resource Allocation: To compute the CPU allocation of each virtual machine we use a proportional-share allocation scheme. However, if we apply the proportional-share policy to allocate to a virtual machine a proportion from the entire cluster, we can reach a situation in which virtual machines cannot be accommodated on the hosts due to their capacity constraints. We illustrate this situation through the following example: we consider 2 hosts each with a capacity of 100 CPU resource units (thus the total cluster capacity is 200 CPU resource units) and 3 virtual machines, each of them having the same bid value b. According to the proportional-share scheme each virtual machine should receive *66.6* CPU resource units. However, the hosts can only accommodate two virtual machines with this capacity. To accommodate all the virtual machines, their allocations need to be adjusted to fit the host capacity constraints. In our example one host accommodates two virtual machines while the other accommodates one virtual machine. Thus, two virtual machines receive a

Parameters	Value	Description
bid_{min}	the reserve price of the provider	The minimum bid the controller can submit
bid_{max}	$\frac{Budget_{remaining}}{time}$	The maximum bid the controller can submit
T_{nvms}	provider limit of virtual machines	Maximum number of virtual machines an application can lease
$alloc_{ref}$	maximum resource utilization	The maximum allocation that is useful for the application
$alloc_{est}$	EWMA(allocation)	Estimated resource allocation for the next time interval. An Exponentially weighted moving average filter is used.
P	provider price	Current resource price
T_{diff}	0.05	Threshold used to avoid too many allocation oscilations
$step$	1	Variable used to decide when to adapt agresively
$direction$	0,-1,1	Variable used to change the scaling direction. A negative value corresponds to a decrease operation while a positive value corresponds to an increase.
$limit$	3	The number of time periods after which the application controllers apply the aggressive policy

TABLE I: Information used by the generic scaling policies.

share of *50* CPU resource units. As we want to keep the proportional-share work-conserving property, the third virtual machine receives *100* CPU resource units.

We obtain this mapping by using a worst-fit decreasing based heuristic. This heuristic sorts the virtual machine requests in descending order by the value of their bid and assigns each request to the host for which the sum of the bids of already assigned requests is minimum. The "ideal" share that each virtual machine should receive is computed by applying the proportional-share scheme on each host.

Load Balancing: As new virtual machine instances need to be created, or the shares for the existing ones change, to ensure that each virtual machine receives its deserved CPU share, some virtual machines need to be migrated from their current hosts to other hosts more fitted for their needs. However, as migration is costly, consuming system resources and affecting the application performance, it is expensive to achieve an optimal mapping of virtual machines to physical hosts (i.e., the mapping that ensures that each virtual machine receives its fair share) at each scheduling interval. There might be cases in which the difference between the actual and ideal allocation for some virtual machines is too small and migrating them to other hosts is useless. Thus, after computing the ideal virtual machine shares, we apply a load-balancing algorithm that tries to reduce the difference between the actual and the ideal allocation (i.e., the allocation error) of each virtual machine while considering the migration cost.

Figure 2 describes the algorithm. The algorithm takes as input the current cluster configuration. Then, at each iteration it tries to accommodate the requirements of the virtual machines that have a negative allocation error (i.e., the ideal allocation is greater than the actual allocation). For this, it first considers the most overloaded hosts. To ensure that the more valuable virtual machines are less migrated, the less valuable virtual machines are the first ones selected for migration. To choose the destination host, a worst-fit heuristic is applied. The solution is improved if the standard deviation over all virtual machine allocation errors is reduced. To memorize the last migration decisions, a tabu list of size 20 is used. The algorithm terminates if the standard deviation becomes smaller than a given threshold or if there is no improvement in the solution for the last 10 iterations. To allow the infrastructure administrator to make a tradeoff between the allocation accuracy and the migration overhead, the algorithm also stops if a maximum number of virtual machine migrations is reached. Then the mapping of virtual machines to physical nodes is sent to the Virtual Infrastructure Manager, indicating which virtual machines should be migrated and created and the CPU share for each of them.

C. Application Control

To ensure that user's SLOs are satisfied, despite resource price dynamicity and application's varying demand, each application is managed by an application controller that scales the resource demand according to application's requirements. To ease the application controller design, we define two classes of scaling policies: (i) vertical and (ii) horizontal. The policies receive as input the performance metric from the application specific sensor and compute the bid and the number of virtual machines for the next time interval. The vertical policies are used to adapt the submitted bid to scale the allocation of a virtual machine. These policies can be used by applications with time-varying resource utilization (e.g., web applications) or rigid applications for which users are willing to make performance-cost tradeoffs (e.g., pay as much as to ensure a specific application progress per time interval). The horizontal policies are used to adapt the number of provisioned virtual machines. These policies can by used by applications that can take advantage of a variable number of virtual machines (e.g., master-worker applications that need to process large amounts of data). For each policy class we define a generic policy that can be extended according to the application's type and SLO.

Figure 3 describes the defined policies and Table I summarizes the information used by them. Both policies can adjust the application bid between a lower (bid_{min}) and an upper (bid_{max}) limit. The lower limit is represented by the reserve price of the provider (we assume that there is a minimum price the users are charged). The upper limit is given by the remaining application budget distributed over a time interval decided according to the application type and SLO.

Vertical Scaling: The vertical scaling policy increases the bid if the actual performance metric, *v*, received from the application is smaller than a reference metric, v_{ref}, and otherwise it decreases it (Lines 1-20, Figure 3). The bid is increased/decreased proportionally to the difference between the actual and reference metric, *T*. The scaling direction is given by the variable *direction*. To ensure a faster convergence to the desired value, we combine this policy with a more aggressive increase triggered if the difference between the actual and the reference metric is maintained over a high time period (Lines 16-19, Figure 3). The moment of triggering the aggressive policy is controlled by the variable *limit*.

Horizontal Scaling: The horizontal scaling policy increases the number of virtual machines if the performance metric of the application is below a given lower limit v_{reflow} and it decreases the number of virtual machines if the performance metric is above an upper limit $v_{refhigh}$ (Lines 22-42, Figure 3). As it is difficult to map the difference between the reference and the actual metric to the number of provisioned virtual machines, we scale the virtual machine number based on two thresholds. The use of two thresholds avoids having too many oscilations in the number of virtual machines. Because booting new virtual machines is costly and decisions to reconfigure the application should be taken carefully, the algorithm adjusts the virtual machine number additively. After deciding the desired number of virtual machines, to ensure that the provisioned virtual machines receive a maximum allocation, the application controller adjusts its bid (Line 36, Figure 3). To avoid being charged with a cost higher than the affordable budget, the number of virtual machines is limited at the maximum number the application controller can afford, $nvms_{max}$.

```
1:  Scale-One-VM(v, v_ref)
2:      if v < v_ref and direction ≠ 1 then
3:          direction = 1
4:          step = 1
5:      if v > v_ref and direction ≠ −1 then
6:          direction = -1
7:          step = 1
8:      step = step + 1
9:      T = |(v_ref − v)/v_ref|
10:     if T < T_diff then
11:         return bid
12:     if step < limit and T < 2 then
13:         factor = 1 + T
14:     else
15:         factor = 2
16:     if direction > 0 then
17:         bid = min{factor · bid, bid_max}
18:     else
19:         bid = max{bid/factor, bid_min}
20:     return bid
21:
22: Scale-Multi-VMs(v, v_reflow, v_refhigh)
23:     if  v < v_reflow and direction ≠ 1 then
24:         direction = 1
25:         step = 1
26:         factor = 1
27:     if v > v_refhigh and direction ≠ −1 then
28:         direction = -1
29:         step = 1
30:         factor = 1
31:     step = step +1
32:     if step < limit  then
33:         factor = factor + 1
34:     else
35:         factor = 2 ∗ factor
36:     bid = alloc_ref · P
37:     nvms_max = min(⌊bid_max/bid⌋, T_nvms)
38:     if direction > 0 then
39:         nvms = min(nvms + factor, nvms_max)
40:     else
41:         nvms = max(nvms − factor, 1)
42:     return (bid, nvms)
```

Fig. 3: Generic scaling policies.

IV. Validation

To show the efficiency and flexibility of our approach we implemented our framework in CloudSim toolkit [4]. We illustrate the use of our framework by considering the case of executing applications under given time constraints (deadlines). We simulate both rigid and elastic applications. Unless otherwise stated, all applications have assigned deadlines. We analyze how our system provides SLO support to applications and how efficient our spot-market model is. To ensure that the policy is usable, we measure the cost in term of virtual machine operations. Finally we show how our approach allows both horizontal and vertical scaling leading to a better resource utilization. We describe next the use of our framework together with the simulation setup and our results.

A. Scaling Examples

We consider the application types defined in Section II, rigid and elastic, and we design two types of application controllers. For the rigid application we design an application controller that estimates the application execution time and by using a vertical scaling policy it keeps the execution time below the user's deadline. The application controller estimates the application execution time by using information about the progress the application makes in its computation. This progress can be defined as a number of iterations the application processes per time interval. We assume the application controller has access to application logs to read periodically the remaining number of iterations and to compute the application progress. The user is also required to submit information regarding the total computation amount needed for application to finish its execution and a performance model to associate the progress rate (i.e., progress made per time period) with the CPU allocation. For the elastic application, we design an application controller that estimates the application execution time and uses a horizontal scaling policy to scale the number of workers to keep the execution time below the user's deadline. The application controller communicates with the application master to retrieve information regarding the remaining number of tasks and the task execution time. The

1: **Vertical-Deadline()**
2: p_{est} = get application predicted progress
3: $p_{ref} = \frac{lenght_{total} - lenght_{current}}{deadline - now}$
4: **if** $alloc_{est} > alloc_{ref}$ **then**
5: $bid = max\{bid/2, bid_{min}\}$
6: **else**
7: bid = $Scale - One - VM(bid, p_{est}, p_{ref})$
8:
9: **Horizontal-Deadline**()
10: $(ntasks, T_{avg}, T_{max})$ = get task execution information
11: $execution\_time = \lceil \frac{(ntasks-1) \cdot T_{avg}}{nvms} \rceil + T_{max}$
12: $execution\_time_{ref} = deadline - now$
13: (bid, nvms)=Scale-Multi-VMs($\frac{execution\_time_{ref}}{execution\_time}$,0.7,0.9)

Fig. 4: Specific derived policies.

control logic behind the policies applied by the application controllers is described in Figure 4.

Scaling a rigid application: To meet its deadline, a rigid application needs to make progress in its computation at a certain rate. For this, the application controller adapts the bids of the virtual machines to keep the progress rate as close as possible to a predefined reference. This reference is computed as the remaining application length (i.e., the total amount of computation) distributed over the remaining time to deadline (Line 3, Figure 4). Based on these metrics, the application controller applies the vertical scaling policy to adapt the virtual machine bids. To avoid spending credits with no benefit, if the application already has a maximum allocation the bid is decreased (Lines 4-5, Figure 4).

When the price is too high to allow continuing the application execution (i.e., the bid the application controller is required to submit for the next time period is bigger than b_{max}) at the current user's budget, the controller also applies a suspend/resume policy. This policy is as following: if the current and predicted progress become smaller than the reference progress the virtual machine instances are suspended and are resumed when the price drops and enough time has passed.

If the current price is too high when the application is submitted, the controller postpones the application execution for a time interval computed as a part of the time that remains between the deadline and the ideal application execution time. If the execution is postponed too much (the remaining time to deadline becomes smaller than the ideal application execution time) the application controller aborts the application. Otherwise, the application controller starts the application by submitting an initial bid computed to get enough allocation to meet the application reference progress at the estimated resource price.

Scaling a master-worker framework: To meet its deadline, the elastic application needs to have enough workers to finish its tasks in time. To estimate the application execution time, we use the result from [17] which establishes the performance bounds for the application execution, given the set of workers and the remaining tasks with their average and maximum execution times (Line 11, Figure 4). Based on these metrics, the application controller applies the horizontal scaling policy to adapt the number of virtual machines/workers. If the estimated execution time is greater than the remaining time to deadline the application controller provisions more virtual machines, and otherwise releases them (Line 13, Figure 4).

These policies can be easily adapted to other SLOs. For example, if an user wants to execute its application as fast as possible, is enough to replace the progress metric with the allocation that the application receives. Thus, the user ensures that its application gets a maximum allocation possible under its given budget constraint.

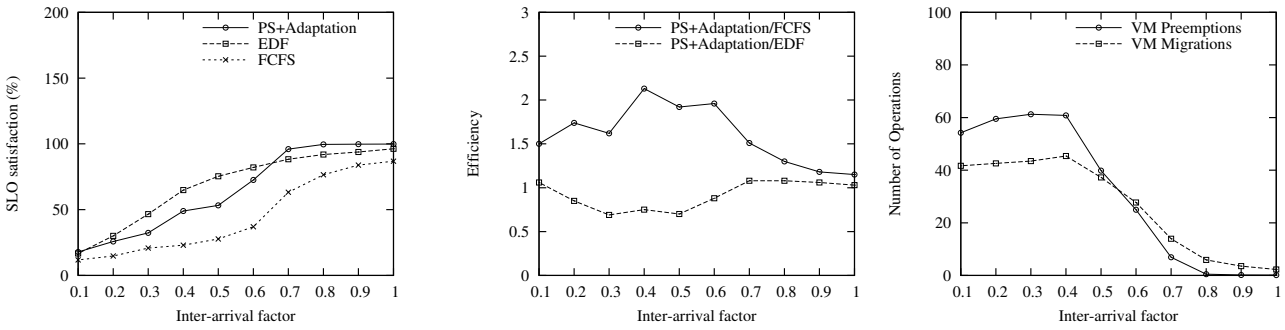
B. System Modeling

When simulating our system we pay attention to two issues: (i) simulating the virtual machine operations for the provider; (ii) generating realistic budgets for applications. For the infrastructure provider, we pay attention to the migration operations, as they have an impact on the application performance. We don't model the performance interference or other overheads, as they would appear for other classic algorithms too (e.g., FCFS or other algorithms employed by current cloud resource managers). We model the migration performance overhead as 10% of CPU capacity used by the virtual machines in which the application is running. This is a high estimation, as previous work found that migration incurs an overhead of approximately 8% on the performance of HPC applications [11]. We estimate the time to migrate the virtual machines as the time to transfer the memory state of the application by using the available network bandwidth.

We generate budgets for application execution requests based on the deadline values. Each application receives a budget computed as $B = B_{base}/deadline$, where B_{base} is the same initial budget assigned to all applications. This is a realistic way to model the fact that users with stringent deadlines value more their applications and are also more inclined to give a higher budget to them. In all cases the scheduling interval is set to 5 minutes.

C. SLO Satisfaction and Efficiency

To measure the total SLO satisfaction that our mechanisms can provide, we simulate traces of real workloads, to which we assign synthetic budgets and deadlines. We use two user-centric metrics: the percentage of met SLOs (deadlines) and the efficiency of our mechanism. The last metric gives us the level of satisfaction the users achieve from using our spot-market system, as compared to non-market based systems. To use this metric, we define the value the application brings to the user as the assigned budget if the application finishes before its deadline, and zero otherwise. We consider that users assign to their applications budgets that reflect their true importance: a user with an urgent application assigns a high budget, and thus will gain more value from executing its application, while a user with a best-effort application assigns a low budget. The efficiency of our mechanism is then



(a) QoS statistics (b) Efficiency (c) Number of Operations

Fig. 5: Application execution statistics.

computed as the ratio between its total provided value and the total value provided by a non-market based mechanism.

Specifically, we compare our mechanism with two well-known, non-market based policies: First-Come-First-Served and EDF (earliest deadline first). We consider the first model as best-effort provisioning, while the second model is QoS-aware. The first model is commonly applied by cloud resource managers. The second model aims to minimize the number of missed deadlines. We use the second model only for comparison reasons. It is not practical for cloud managers to apply this model. If they did, they would support only a single type of application goal (i.e., meeting deadlines).

We simulate a real-world workload of batch applications, taken from an online archive [2] on an infrastructure of 240 hosts. This archive contains public logs of HPC resource management systems. These logs have information regarding the resource demand and execution time of applications submitted to the infrastructure. We used the HPC2N log, as it has detailed information about application memory requirements, needed to estimate the migration overhead. From this trace we selected the first 1000 applications. The applications are assigned synthetic deadlines between 1.2 and 10 times their ideal execution times. The assigned budget is $B_{base} = 60$ credits/resource unit/scheduling period, renewed at a rate of 20 credits/resource unit/scheduling period. To simulate different utilization levels, we vary the application inter-arrival time (i.e., the time between two consecutive submissions), by multiplying it with a factor (i.e., inter-arrival factor).

Figure 5(a) shows the number of applications that meet their SLOs. In lightly-loaded periods, our mechanism outperforms both EDF and FCFS. When the system becomes overloaded, the mechanism continues to outperform FCFS, but not EDF. Similar results are achieved for the efficiency metric, as illustrated in Figure 5(b). Our mechanism is 2 times more efficient than FCFS in high load periods while it encounters an efficiency loss of up to 25% compared to EDF.

The performance gap between our mechanism and EDF can be explained as follows. The nature of our mechanism is decentralized: there is no centralized entity that decides the allocation of every application to meet a system-wide objective (e.g., minimizing missed deadlines). Rather, each controller acts selfishly and independently from each other to meet its own application SLO. Thus, the solution is less optimal than a centralized solution. However, the main advantage of our solution comes exactly from this decentralization: as each controller is independent, we can support arbitrary application types and SLOs compared to a centralized mechanism that strives to accomplish a fixed global objective. This experiment shows that our mechanism provides good SLO satisfaction to applications and achieves a good efficiency. The performance degradation is the "price" payed by the decentralized nature of our system, that allows each user to behave in a strategic selfish way.

D. Policy Cost

With the same settings from the previous experiment, we measure the cost of our mechanism in term of virtual machine operations. We log the number of migrations and suspend/resume operations for each time interval during the experiment run. Figure 5(c) shows the average number of operations per time interval for each experiment run. Given the large scale of the infrastructure these numbers are reasonable: a maximum of 61 preempted virtual machines and 45 migrations when the system becomes highly loaded. This proves that our system can be used in a real environment, as the virtual machine operations would not have a highly negative impact on the application performance.

E. Flexibility

To prove that our system is not build specifically for one application type, we show the good behavior of our horizontal scaling policies also. For this, we consider an elastic master-worker application that applies the horizontal policy to manage a burst in its workload and finish its execution before the deadline given by the user. The application has 1075 tasks to process in an time interval of 643.25 minutes, with each task having an uniform execution time between 1 and 40 minutes. After the first *85* minutes of execution, an additional workload of 1500 tasks is generated.

Figure 6 shows the number of virtual machines provisioned by our policy in time (expressed, for simplicity, in scheduling

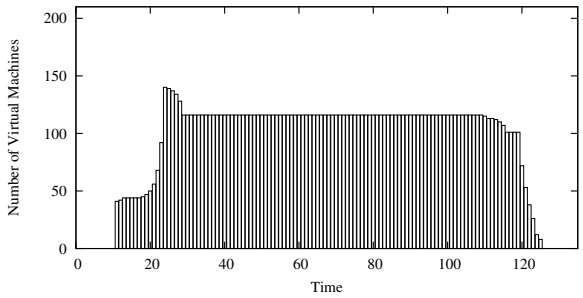


Fig. 6: Application elastic scaling.

intervals). After starting its execution the application controller provisions aproximately 44 virtual machines, as this number is enough for it to finish its execution in a timely manner. When the additional workload is submitted, the application controller starts provisioning more virtual machines to compensate for the increase in demand. The number is increased first incrementally, and then more and more aggressively. In this case, the controller overestimates the number of provisioned resources, so it decreases it afterwards, reaching an equillibrum point at 116 virtual machines. When the application is close to finishing its execution, less tasks remain to be processed and so the controller shuts down the idle virtual machines. This experiment shows that our mechanism allows applications to react reasonably fast to changes in their demand to meet their SLOs. We consider that for HPC workloads these changes don't need to be very fast, so simple policies would suffice.

V. Conclusion

In this paper we proposed Themis, a system that shares the resources among applications using a spot market while providing support for application SLOs. Our system is motivated by the current cloud provisioning models, which provide "on-demand" virtualized resources. In contrast with classic resource management systems, clouds provide more flexibility to users in meeting their SLOs. However the provider has to support the burden of managing resources efficiently. To address this problem, Themis brings the following contributions: (i) an implementation of a proportional-share auction that maximizes resource utilization while considering virtual machine migration costs; (ii) a set of feedback-based control policies used to adapt the application bid and resource demand to fluctuations in price.

Our experiments have shown how Themis is capable of providing overall good SLO support. The decentralized nature of markets makes our system flexible to use by different application types. We have shown how applications with dynamic changes in their demand can make use of our system, thus demonstrating that Themis is not designed for only one application type but it provides generic interfaces that can be used by any application.

Currently we tested Themis through simulations. We also plan to evaluate Themis on a real testbed. As next steps, we plan to improve our resource allocation mechanism by supporting multiple resource types. We also plan to extend our policies for more application types and SLOs. For example, adaptive applications could scale their resource demands to perform computation at a certain progress rate. Another concern that we plan to address is how to allow users to specify placement preferences for their applications. Such preferences are useful for improving application performance or for fault tolerance.

References

[1] AmazonEC2. http://aws.amazon.com/ec2/.

[2] P. W. Archive. http://www.cs.huji.ac.il/labs/parallel/workload/.

[3] A. AuYoung, B. N. Chun, A. C. Snoeren, and A. Vahdat. *Resource Allocation in Federated Distributed Computing Infrastructures*, pages 1–10. 2004.

[4] R. N. Calheiros, R. Ranjan, A. Beloglazov, C. A. F. De Rose, and R. Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[5] B. N. Chun, P. Buonadonna, A. AuYoung, D. C. Parkes, J. Shneidman, A. C. Snoeren, and A. Vahdat. Mirage: A microeconomic resource allocation system for sensornet testbeds. *The Second IEEE Workshop on Embedded Networked Sensors 2005 EmNetSII*, pages 19–28, 2005.

[6] M. Feldman, K. Lai, and L. Zhang. The proportional-share allocation market for computational resources. *IEEE Transactions on Parallel and Distributed Systems*, 20(8):1075–1088, 2009.

[7] A. S. Instances. http://aws.amazon.com/ec2/spot-instances/.

[8] K. Lai, L. Rasmusson, E. Adar, L. Zhang, and B. Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent and Grid Systems*, 1(3):169–182, 2005.

[9] M. Mattess, C. Vecchiola, and R. Buyya. Managing peak loads by leasing cloud infrastructure services from a spot market. In *12th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 180–188. IEEE, 2010.

[10] M. Mazzucco and M. Dumas. Achieving performance and availability guarantees with spot instances. *13th International Conference on High Performance Computing and Communications (HPCC)*, pages 296–303, 2011.

[11] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive fault tolerance for hpc with xen virtualization. *Proceedings of the 21st annual international conference on Supercomputing (ICS)*, page 23, 2007.

[12] T. Sandholm and K. Lai. A statistical approach to risk mitigation in computational markets. In *Proceedings of the 16th international symposium on High performance distributed computing*, HPDC '07, New York, NY, USA, 2007. ACM.

[13] T. Sandholm and K. Lai. Dynamic proportional share scheduling in hadoop. In *15th Workshop on Job Scheduling Strategies for Parallel Processing*, 2010.

[14] C. Shin Yeo and R. Buyya. Pricing for utility-driven resource management and allocation in clusters. *International Journal of High Performance Computing Applications*, 21(4):405–418, 2007.

[15] B. Sotomayor, R. Montero, I. Llorente, and I. Foster. An Open Source Solution for Virtual Infrastructure Management in Private and Hybrid Clouds. *IEEE Internet Computing*, 13(5):14–22, 2009.

[16] P. B. System. http://www.pbsworks.com/.

[17] A. Verma, L. Cherkasova, and R. H. Campbell. Aria : Automatic resource inference and allocation for mapreduce environments. *HPL201158*, pages 263–275, 2011.

[18] S. Yi, D. Kondo, and A. Andrzejak. Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud. *IEEE 3rd International Conference on Cloud Computing*, pages 236–243, 2010.

[19] S. Zaman and D. Grosu. Combinatorial auction-based allocation of virtual machine instances in clouds. In *IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 127–134. IEEE, 2010.